LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-555978

MPI Runtime Error Detection with MUST: Advances in Deadlock Detection

T. Hilbrich, J. Protze, M. Schulz, B. de Supinski, M. Mueller

May 7, 2012

SC2012
Salt Lake City, UT, United States
November 10, 2012 through November 16, 2012

Disclaimer

MPI Runtime Error Detection with MUST: Advances in Deadlock Detection

Tobias Hilbrich*, Joachim Protze*, Martin Schulz†, Bronis R. de Supinski† and Matthias S. Müller*
*Technische Universität Dresden, ZIH
D-01062 Dresden, Germany
Email: {tobias.hilbrich, joachim.protze, matthias.mueller}@tu-dresden.de
†Lawrence Livermore National Laboratory
Livermore, CA 94551
Email: {bronis, schulzm}@llnl.gov

***Abstract*—The widely used Message Passing Interface (MPI) is complex and rich. As a result, application developers require automated tools to avoid and to detect MPI programming errors. We present the Marmot Umpire Scalable Tool (MUST) that detects such errors with a significantly increased scalability. We present improvements to our graph-based deadlock detection approach for MPI, which cover complex MPI constructs, as well as future MPI extensions. Further, our enhancements check complex MPI constructs that no previous graph-based detection approach handled correctly. Finally, we present optimizations for the processing of MPI operations that reduce runtime deadlock detection overheads. Existing approaches could require $O(p)$ analysis time per MPI operation, for p processes, where our improvements lead to an $O(\log p)$ complexity or better for real world applications. We present overhead measurements for two major benchmark suites with up to 1024 cores to demonstrate our improvements for real world scenarios.**

I. Introduction

The Message Passing Interface (MPI) [1] is a de facto standard for parallel programming. It provides a comprehensive API that enables users to efficiently and portably exchange messages between processes. The standard's design targets and enables high performance through low latency communication and high scalability, but provides few syntactic or semantic extensions to enforce its correct use.

As a result, MPI applications can exhibit a wide range of error classes. Simple errors result from invalid arguments such as an invalid array length specification, while other errors may involve MPI resources such as communicators or requests, e.g., the user starts a nonblocking communication that he or she does not complete before calling `MPI_Finalize`. Such errors are often hard to detect, since root-cause and symptoms are far apart. Further, some MPI usage errors may only occur for particular interleavings, for some MPI implementations, or for some systems. Examples for the latter result from freedoms in the MPI standard that allow, but do not require, implementations to buffer point-to-point communications or to make collective calls synchronizing.

Various runtime error detection tools exist that can detect such errors. They can detect some errors directly on the application processes as a local correctness check, e.g., the use of an invalid MPI datatype. Whereas other errors such as type mismatches in message data or messaging deadlocks require information about more than one process, and hence have to be implemented using a non-local approach. Thus, these runtime tools need to communicate information from the application processes to a process or thread that runs these non-local correctness checks, which complicates the design and scalability of these tools. As a consequence, current tools are either incomplete in their functionality or scale poorly, which renders them insufficient for reliably detecting errors in large scale applications.

In this paper, we present the runtime tool MUST (Marmot Umpire Scalable Tool, named after its predecessor tools) that aims at overcoming these shortfalls of current tools and at providing a scalable solution for efficient MPI error checking at runtime. MUST detects many classes of MPI correctness errors and is built on top of a flexible plugin concept, that allows users to customize the tool to the error classes of interest, as well as to extend its functionality to cover new classes of errors. Although, MUST covers various process-local correctness checks, for this paper we focus on its non-local checks, primarily deadlock detection. Specifically, we describe MUST as it can be used to easily and correctly detect, analyze, and guide the removal of deadlocks in MPI application.

Fig. 1 sketches correct and incorrect MPI communications. We use the notation "Recv(from:x)" for an `MPI_Recv` call that uses `MPI_COMM_WORLD` as communicator, x as the argument specifying the source rank, while the remaining arguments are removed for sake of simplicity in the example. Similarly, we use "Send(to:x)" for `MPI_Send` and "Barrier()" for `MPI_Barrier`, respectively. Fig. 1(a) shows a correct communication between two processes, whereas examples 1(b) and 1(c) are erroneous variations of this example. In Fig. 1(b) both processes will block in the receive call and wait for each other to issue a matching send call. This will never happen, as both processes are blocked (assuming no additional application threads execute MPI calls). This example will always deadlock, whereas Fig. 1(c) shows an implementation dependent deadlock. In MPI, calls to `MPI_Send` are usually buffered for small messages, which would allow both tasks to invoke their send calls and then their receive calls. This example deadlocks, however, if the send calls are not buffered

Process 0	Process 1
Send(to:1)	Recv(from:0)
Recv(from:1)	Send(to:0)

(a) Send-recv communication.

Process 0	Process 1
Recv(from:1)	Recv(from:0)
Send(to:1)	Send(to:0)

(b) Recv-recv deadlock.

Process 0	Process 1
Send(to:1)	Send(to:0)
Recv(from:1)	Recv(from:0)

(c) Send-send deadlock.

Process 0	Process 1	Process 2
Send(to:1)	Recv(from:ANY)	Send(to:1)
	Recv(from:2)	
Barrier()	Barrier()	Barrier()

(d) Schedule-dependent deadlock.

Process 0 (Master)	Processes 1 to $(\text{size} - 1)$
for $(i = 0;\ i < \text{size};\ i++)$	Send(to:0)
Recv(from:ANY)	

(e) Master-slave communication.

Fig. 1. MPI communication examples.

and is incorrect as a result.

Fig. 1(d) presents a potential deadlock that only manifests itself in some runs. The *Recv(from:ANY)* call of process 1 uses a so called wildcard source (`MPI_ANY_SOURCE` in MPI) that allows the receive operation to match a message from any process. If this call receives the message from process 0, the second receive of process 1 can receive the message from process 2. All three processes can then complete the *Barrier* call and continue their execution. Alternatively, if the first receive of process 1 receives the message from process 2, then the second receive of process 1 cannot complete, as process 2 does not send another message before it issues the *Barrier* call. Since process 1 cannot issue the *Barrier* until process 2 sends the message, both processes block indefinitely. These wildcard receives, as well as other MPI constructs, can lead to interleaving dependent MPI deadlocks, which only occur in some application runs. MPI deadlock detection tools must pay particular attention to handle these constructs correctly.

We follow a graph-based approach that uses the AND⊕OR model [2] to represent wait-for dependencies of active MPI calls. This model simplifies the AND-OR model, which can model wait-for conditions of the most general type. Our approach uses a graph analysis in the AND⊕OR model to recognize and visualize deadlocks. We present a generalization to this model, as well as improvements in its application for runtime error detection tools.

Specifically, our contributions include:

- Generalization of the AND⊕OR model to handle the full range of MPI usage scenarios previously only covered by the more complex AND-OR model;
- Optimizations in runtime deadlock detection that analyze most MPI operations with $O(\log p)$ complexity for p processes;
- Extension to detect deadlocks that involve nonblocking wildcard receives robustly in complex applications; and
- A comprehensive application study that highlights the benefits of our optimizations.

We structure the rest of this paper as follows: Section II presents related work and compares our graph-based approach; Afterwards, we introduce the AND⊕OR model and detail our runtime error detection tool MUST (Section III); We present our generalization of the AND⊕OR model in Section IV and highlight the architectural and operational changes to the existing deadlock detection approach that support a far more efficient analysis of MPI operations in Section V; Section VI presents the challenges and technology that surround the handling of nonblocking wildcard receives; We demonstrate the results of our improvements with two benchmark suites in Section VII.

	Recv-recv (Fig. 1(b))	*Send-send* (Fig. 1(c))	*Schedule-dependent* (Fig. 1(d))
Timeout	Yes	No	Run
ISP	Yes	Yes	Yes
DAMPI	Yes	No	Yes
AND⊕OR	Yes	Yes	Run

Fig. 2. Runtime approach comparison.

II. Related Work

Our work is closely related to MPI runtime error detection tools such as ISP [3], MPI-Check [4], MPICH extension [5], Marmot [6], and Umpire [7]. Fig. 2 compares these approaches for the deadlock examples from Fig. 1, where we assume that the MPI implementation buffers the *Send* call in 1(c), i.e., the example runs without producing a deadlock.

The MPICH extension ignores deadlocks and focuses on other correctness checks, whereas Marmot and MPI-Check both implement a timeout-based MPI deadlock detection. The timeout approach detects the recv-recv deadlock (1(b)), but will not detect the send-send deadlock (1(c)), and the schedule-dependent deadlock (1(d)) only if the error manifests (denoted by *Run*). Further, a timeout approach can lead to false positives and, further, does not provide a graphical representation for the source of the deadlock.

ISP investigates all alternative interleavings of send/recv pairs in an MPI application to verify deadlock freedom for nondeterministic MPI programs. As a result, ISP analyses both execution paths of the schedule-dependent deadlock example

and always detects this error. ISP uses a centralized scheduler to play through all interleavings by re-executing the application multiple times, and terminates after investigating all possible interleavings. Although this provides the best coverage, several communication patterns can produce an exponential number of different interleavings. Fig. 1(e) presents such an example in which ISP will not be able to validate all of the example's possible interleavings, even for a few tasks. Thus, even though ISP provides good coverage, the applicability of the tool can be limited.

The DAMPI [8] approach uses a distributed detection of alternative interlavings to overcome the limitations of ISP. To explore different interleavings, DAMPI rewrites MPI calls based on an existing enumeration of explorations to cover. This effectively removes ISP's centralized scheduler. For each interleaving, DAMPI executes the application and detects deadlocks with a timeout. In summary, this approach can both detect the recv-recv and the schedule-dependent deadlock, but DAMPI will not catch the send-send deadlock, due to the timeout-based deadlock detection. Also DAMPI may give false positives and lacks a visualization of deadlocks as a result.

Umpire uses the AND⊕OR model with a graph-based deadlock detection that can detect both the recv-recv and the send-send deadlock. The approach will only detect the schedule-dependent deadlock if the error manifests. This approach can't lead to false positives and provides a graphical representation of the source of the deadlock. We build on this model and add support for MPI scenarios that the Umpire deadlock conditions do not cover and implement several significant optimizations to that approach. Further, this approach could be used to replace the timeout approach in DAMPI to overcome the drawbacks of timeout-based deadlock detection.

Finally, our generalization of the AND⊕OR model [2] extends work on the AND-OR graph-theoretic deadlock model [9], [10], [11]. Visualizations of the AND-OR model, where processes may specify a Boolean equation as their wait-for condition, are difficult to visualize, while our work demonstrates that the AND⊕OR model is equivalent and supports intuitive visualization. Our transformation of the AND-OR model suits the limited AND-OR semantics of MPI but might lead to many additional nodes for other uses of AND-OR dependencies. Thus, its efficacy for general deadlock scenarios (i.e., beyond MPI) remains an open question.

III. MUST and the AND⊕OR Model

In order to be valuable to the end user, we require a deadlock detection that provides no false positives and enables a comprehensive understanding of the source of the deadlock. Such a solution must also be applicable to all MPI applications with an acceptable overhead. Experience with ISP's exploration of all alternative interleavings shows that such a thorough analysis can be impractical at scale. Thus, we restrict our approach to only consider deadlocks that manifest themselves in a given run. As a result, we use the AND⊕OR model [2] that enables a graph-based deadlock detection, to implement a runtime deadlock detection approach in the Marmot Umpire Scalable Tool (MUST), which extends and scales the functionality of its predecessors: Marmot [6] and Umpire [7].

A. Runtime Deadlock Detection with MUST

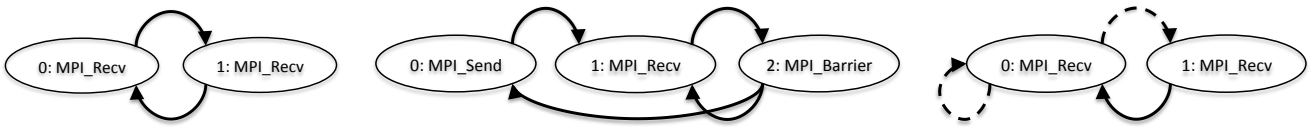
We attach the MUST library to all application processes to intercept their MPI calls at runtime. The tool then checks their correctness either directly on the application processes, or within additional tool nodes of a tree based overlay network. The tree network allows scalable data aggregation and can run distributed or centralized correctness checks. For the purpose of deadlock detection, we currently use a centralized *deadlock detector* that runs the graph-based deadlock detection on the root of the tree network. Each application process forwards information about communication calls to the centralized deadlock detector that analyzes all communication calls.

The detector tracks the state of collective operations, as well as queues for outstanding point-to-point communications. With that, MUST can evaluate whether a certain MPI call can complete or whether the call waits for another communication call, e.g., a point-to-point communication that matches the call. We use this information to capture all wait-for dependencies between the processes. We represent these wait-for conditions as a graph and use a graph analysis to determine whether a deadlock exists at a certain execution step of the application.

B. The AND⊕OR Model

MUST's detector evaluates each MPI operation to determine whether that operation can complete, or whether it waits for one or multiple processes to invoke required MPI operations. In the simplest case, an MPI process waits for exactly one other process, e.g., a blocked `MPI_Ssend` call that has no matching receive. More complex dependencies can arise from MPI usage, e.g., a process issues an `MPI_Barrier` call and waits for all processes in the communicator that have not yet issued a matching call. A different behavior holds for processes that issue an `MPI_Recv` call with the wildcard source, `MPI_ANY_SOURCE`. The process that issues the call waits for all processes in the communicator until any *one* of them issues a matching send. In summary, MPI processes can wait either for all or for any one process in a subset of `MPI_COMM_WORLD`.

We use the terms *AND* semantics for processes that wait for all processes of a process-set and the term *OR* semantics for processes that wait for any process of a process-set. Deadlock criteria exist for both the *AND* semantics (a cycle) and the *OR* semantics (a knot, i.e., A set of nodes X where each node has X as descendant set). MPI usage can mix both semantic types and hence we need a more general model. The most general deadlock model, the AND-OR model, which allows arbitrary combinations of *AND* or *OR* conditions, is sufficiently general but more so than necessary. This model's generality makes analysis harder and graphical visualizations impractical. We therefore use the AND⊕OR model, which limits each node in the graph to either exclusively use *AND* or *OR* semantic arcs. Its wait-for graph (WFG) [2] uses the following definition:



(a) WFG for Fig. 1(b). (b) WFG for Fig. 1(d) (c) WFG for wildcard receive DL.

Fig. 3. AND⊕OR WFG examples.

Process 0	Process 1	Process 2	Process 3
Irecv(comm:A, from:ANY, &reqs[0])	Recv(from:2)	Recv(from:0)	Recv(from:2)
Irecv(comm:B, from:ANY, &reqs[1])			
Waitall(2, reqs)			

Fig. 4. AND-OR semantics in MPI (Communicator A contains tasks 0, 1, and 2, communicator B contains tasks 0, 2, and 3).

Definition 1 (AND⊕OR WFG): A tuple $(V, E_{\text{AND}}, E_{\text{OR}})$ forms an AND⊕OR WFG if $(V, E_{\text{AND}} \cup E_{\text{OR}})$ is a directed graph and the following restriction holds:

- $\{v \in V | \exists x \in V : (v, x) \in E_{\text{AND}}\} \cap \{v \in V | \exists x \in V : (v, x) \in E_{\text{OR}}\} = \emptyset$

We represent each process with a node in V and use arcs of two different types to model wait-for dependencies between the processes that we associate with the nodes. The WFG allows each node to use either arcs of the *AND* semantics (arcs in E_{AND}) or the *OR* semantics (arcs in E_{OR}). Fig. 3(a) shows the WFG for the deadlock from Fig. 1(b). This graph only uses the *AND* semantic that we illustrate with solid arcs. Fig. 3(b) shows the WFG for the deadlock from Fig. 1(d) for the interleaving that leads to deadlock. We assume that process 0 is blocked in the `MPI_Send` call, as MUST treats standard mode sends as unbuffered. Finally Fig. 3(c) shows a WFG with the OR semantics, for which we use dashed arcs. The graph represents the wait-for conditions of a variation of example 1(b), where process 0 uses a wildcard source instead.

We use the OR-knot [2] as a deadlock criterion for AND⊕OR WFGs: A set of nodes X where each node can reach all nodes in X and for which no node has an outgoing arc of the *OR* semantics that leads to a node that is not in X. MUST implements a graph search to detect and visualize this criterion. This allows us to distinguish the root of a deadlock from other processes that are waiting for deadlocked processes.

In the following sections we address and provide new solutions for three fundamental limits in the application of the AND⊕OR model for runtime deadlock detection:

- Some MPI constructs that are too general for the AND⊕OR model;
- Overhead in analyzing the wait-for dependencies of MPI calls; and
- Complexities in wildcard receive handling.

IV. AND⊕OR Generalization

In isolation, all blocked MPI processes use either the *AND* or the *OR* semantics. However, combinations of different wait-for scenarios can lead to scenarios in which this property no longer holds. The example in Fig. 4 shows such a case using an `MPI_Waitall` call and two user-defined communicators. The user defined communicator A contains processes 0, 1, and 2, whereas communicator B contains processes 0, 2, and 3. In the example, process 0 blocks until both non-blocking wildcard communication requests are completed, which leads to the following situation: Process 0 waits for both requests to complete (*AND* semantics), which in turn wait for one process out of a set of processes (*OR* semantics). Thus, this example uses the *AND* and the *OR* semantics for a single process, which the AND⊕OR model cannot handle. Further, particular constructs in MPI can also lead to such cases, including the call `MPI_Sendrecv`, multi-threaded MPI applications, and nonblocking collectives [12] as they will appear in MPI-3.

A. Transformation

The AND⊕OR WFG lacks the necessary generality to handle scenarios as in Fig. 4 directly. We could use the AND-OR model for these cases but would lose our graphical deadlock criterion. Thus, we provide a novel transformation for wait-for dependencies of the AND-OR model that adds additional WFG nodes to translate them into the AND⊕OR model. Intuitively, we add additional nodes that separate the wait-for conditions of nodes that both use the *AND* and the *OR* semantic such that in the resulting wait-for graph each process (or node) only uses one of the two semantics. With that we demonstrate the generality of the AND⊕OR model compared to the AND-OR model, while our model still provides the benefits of intuitive visualization for common MPI usage.

In general deadlock theory, wait-for dependencies link processes to resources. In MPI, however, processes wait for messages that only processes can generate. Therefore, we simplify our notation and formulate each wait-for dependency as from a process to a process, without loss of generality. Thus, we define AND-OR wait-for dependencies as follows:

Definition 2 (AND-OR wait-for dependency): For a process set, V, and the associated set of valid wait-for dependencies, Λ, each wait-for dependency $w \in \Lambda$ equals one of the following:

- v with $v \in V$
- $(w_1 \wedge w_2)$ with $w_1, w_2 \in \Lambda$
- $(w_1 \vee w_2)$ with $w_1, w_2 \in \Lambda$

$$t(v,R) = \begin{cases} (\{v,R\},\{(v,R)\},\emptyset) & : R \in V \\ t(x,R_1) \cup t(y,R_2) \cup & \\ (\{v,x,y\},\{(v,x),(v,y)\},\emptyset) & : \begin{array}{l} R = (R_1 \wedge R_2) \\ x = concat(\alpha, v) \\ y = concat(\beta, v) \end{array} \\ t(x,R_1) \cup t(y,R_2) \cup & \\ (\{v,x,y\},\emptyset,\{(v,x),(v,y)\}) & : \begin{array}{l} R = (R_1 \vee R_2) \\ x = concat(\alpha, v) \\ y = concat(\beta, v) \end{array} \end{cases}$$

Fig. 5. Translation of AND-OR dependencies to AND⊕OR dependencies.

Informally, the set Λ equals the set of all Boolean equations that use processes in V as atoms. Each process v can specify a wait-for dependency $R \in \Lambda$, which we denote with a tuple (v, R). We do not require each process to have a wait-for dependency. We translate a set of general wait-for dependencies, $(v_1, R_1), (v_2, R_2), ..., (v_n, R_n)$, into the AND⊕OR model as follows:

$$\text{translate}(((v_1,R_1),(v_2,R_2),...,(v_n,R_n))) \quad = \quad \bigcup_{i=0}^{n} t(v_i, R_i)$$

Fig. 5 shows the function t that we use to translate the wait-for dependency of each process individually. We define the union of AND⊕OR WFGs $(V^1, E^1_{\text{AND}}, E^1_{\text{OR}})$ and $(V^2, E^2_{\text{AND}}, E^2_{\text{OR}})$ as $(V^1 \cup V^2, E^1_{\text{AND}} \cup E^2_{\text{AND}}, E^1_{\text{OR}} \cup E^2_{\text{OR}})$. This union returns an AND⊕OR WFG if and only if each node in $V^1 \cup V^2$ either uses arcs of the *AND* type or arcs of the *OR* type. The union operator combines the intermediate results that function t returns into a final AND⊕OR WFG. The function t matches the recursive Definition 2 of the AND-OR wait-for dependencies and translates the wait-for dependency of a node v with a recursive scheme:

$R \in V$: (a direct dependency) t adds the nodes v (the depending node) and R, to the set of nodes in the WFG and adds an *AND* semantic arc from v to R;

$R = (R_1 \wedge R_2)$: t introduces two additional nodes to the WFG, one prefixed with "α" and one prefixed with "β"; This case returns a WFG that results from the union operation applied to three WFGs: the WFGs obtained with recursive calls to $t(x, R_1)$, which translates the wait-for dependency R_1 applied to the first additional node, and to $t(y, R_2)$, which translates R_2 applied to the second additional node; and the WFG that contains the node v and the two additional nodes x and y along with two *AND* arcs from node v to x and y;

$R = (R_1 \vee R_2)$: Like the previous case, but the arcs from v to x and y have the *OR* semantic.

We assume that both the "α" and the "β" symbol are not used in any node in the process set, V. Thus, t either adds arcs of the *AND* semantic or arcs of the *OR* semantic in each construction step. As t creates a valid AND⊕OR WFG in each step, while each recursive call to t uses a different symbol for the first argument (the depending node), we conclude that t returns an AND⊕OR WFG.

B. Example

If we consider the example from Fig. 4, then process 0 waits for both non-blocking receives to complete. The first receive waits for processes 0, 1, or 2, whereas the second waits for processes 0, 2, or 3. This corresponds to the AND-OR wait-for dependency: $(((0 \vee 1) \vee 2) \wedge (2 \vee (3 \vee 0))$. The remaining three processes are blocked in receive calls where processes 1 and 3 waits for process 2, while process 2 waits for process 0.

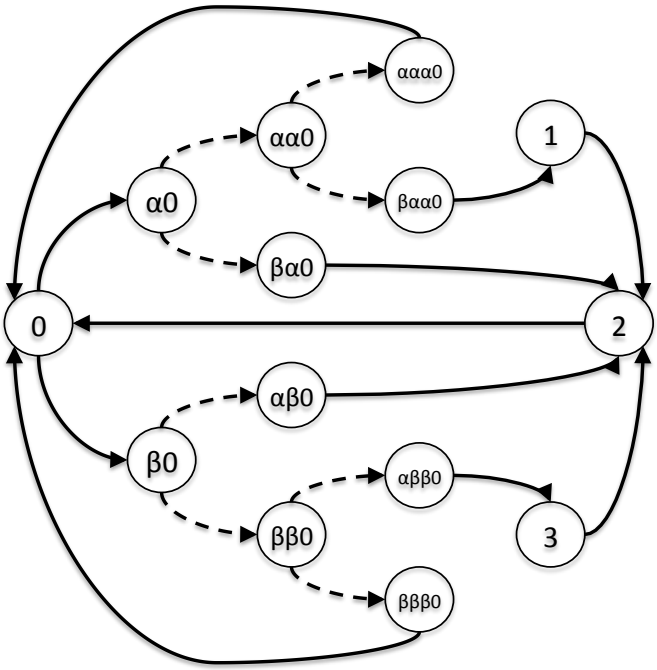
Fig. 6(a) shows the result of applying the $translate$ function to this example. We use solid arcs to illustrate arcs of the *AND* semantic and use dashed arcs for those of the *OR* semantic. The graph shows that the *translate* function first introduces the additional nodes $\alpha 0$ and $\beta 0$ for process 0. The transformation uses $\alpha 0$ to represent $((0 \vee 1) \vee 2)$ of the wait-for condition of process 0 and $\beta 0$ to represent $((2 \vee 3) \vee 0)$ respectively. The WFG uses unnecessary intermediate nodes, e.g., $\alpha\alpha 0$ and $\beta\alpha 0$, which can be avoided with more elaborate transformation functions. Extensions of *translate* in MUST handle conjunctions and disjunctions of arbitrary order (Fig. 6(b)). Also we remove intermediate nodes that have exactly one incoming and one outgoing arc (Fig. 6(c)). Finally, our implementation in MUST replaces the additional node symbols of t with more meaningful labels. Thus, MUST provides the WFG in Fig. 6(d) for the example in Fig. 4. This more intuitive representation highlights that the `MPI_Waitall` call uses two requests, both resulting from calls to `MPI_Irecv`. We use a different node shape—a parallelogram—to indicate that the two additional nodes represent complex MPI call semantics instead of processes.

C. Deadlock Criterion

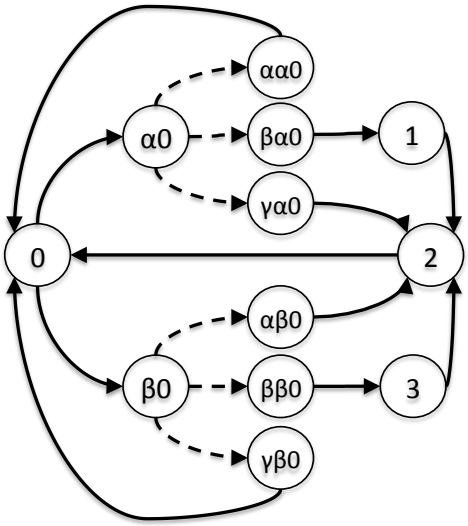
Finally, if *translate* returns a WFG with a deadlock, we can use the OR-knot [2] as a graphical deadlock criterion. Fig. 6(d) shows an example: the nodes that are filled in gray form an OR-knot. Each node in this set can reach all other nodes in the set, while no node has an outgoing *OR* arc that leads to a node not in the set. Note that further OR-knots exists in this example, the first includes all nodes, whereas the second contains processes 0, 2, and 3, as well as the `MPI_Irecv` node for "request[1]".

Even if users of MUST are not interested in the details of the AND⊕OR model, we can provide them a simpler output: A list of processes that form a deadlock. Even further, this list includes our intermediate nodes, so we will not only point users to a certain process that issues an `MPI_Waitall` call, but also to the request(s) that are important.

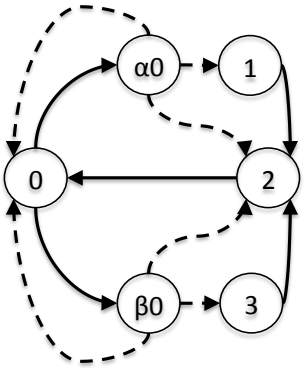
Deadlocks in the AND-OR model relate to deadlocks in the constructed AND⊕OR WFG. If *translate* returns an



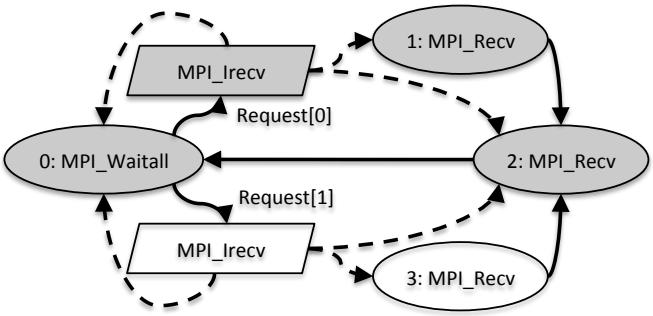
(a) WFG that results from the *translate* function.



(b) Result of *translate* extended to ternary (or higher order) Boolean operators.



(c) Extension of Fig. 6(b) that removes unnecessary intermediate nodes.



(d) MUST output.

Fig. 6. AND⊕OR WFGs for the Example of Fig. 4.

AND⊕OR WFG with an OR-knot, for each node x that was added by t and that an OR-knot includes, the following holds: If x was added by processing a wait-for condition of a process v, then this criterion also includes v itself, since any path that leads to x from another process must pass through v. Further, none of the intermediate nodes can form a cycle (or even an OR-knot) without using arcs of any node that represents a process. Thus, a process is deadlocked if the node that represents this process is part of an OR-knot. However, some of the additional nodes that are derived from a process v may not be part of an OR-knot, as Fig. 6(d) illustrates.

V. Optimized Deadlock Analysis

Our generalization of the AND⊕OR model allows us to model wait-for dependencies caused by any call in the current and upcoming MPI standard. However, this capability alone is insufficient. MUST also has to optimize the processing of MPI operations in order to reduce overhead and interference.

Runtime deadlock detectors intercept MPI calls and interpret them based on the MPI standard to determine which MPI calls are blocked at a certain step of the application run. We need that information to compute the WFG, which we analyze to detect any deadlocks. Current approaches run this analysis on a central process or thread [7], [3]. For our long-term goal of deadlock detection that will scale to 10,000 or more processes, we must make the following processing steps scalable:

- Point-to-point matching;
- Collective matching;
- Wait-state analysis; and
- Graph-based deadlock detection.

The remainder of this section details optimizations of these steps that we implement in MUST.

A. Runtime Detection Costs

Graph-based deadlock detection has a complexity of $O(p^2)$ for p processes [2] and, thus, forms the most expensive processing step. However, MPI semantics allows a deadlock detector to analyze the MPI calls pessimistically: The detector does not need to analyze any MPI call of a process that is currently blocked in a preceding MPI call. Thus, even if a deadlock exists, the detector can process further MPI calls of other processes. As a result, we do not need to run the graph-based deadlock detection when we analyze an MPI event. We invoke the deadlock detection only if we suspect the presence of a deadlock:

- If the detector receives no additional MPI events within a configurable timeframe;
- If only some processes send MPI events; or
- When all processes notify the detector of a call to `MPI_Finalize`.

Thus, we infrequently invoke the graph-based search, where a single search for about 10,000 processes completes within seconds. As a result, the major overhead for our runtime deadlock detection results from the other three processing steps rather than from the graph-based deadlock detection.

As the detector must intercept and analyze more MPI calls at scale, the overhead for point-to-point and collective matching increases. We will eventually need distributed approaches to cope with the increased workload [13] in these steps. MUST still uses centralized components, but based on a drastically improved wait-state analysis. In Umpire [2], the detector tracks the WFG at all times. Consider calls such as wildcard receives or collective calls that introduce $O(p)$ arcs to the WFG for p processes ($(p-1)/2$ arcs on average for collective calls and p arcs on average for wildcard receives). With this approach the overhead to update the WFG for a single operation increases linear with scale. Also, most applications increase their number of communication calls linear with scale. If both effects scale linear, the total overhead for runtime deadlock detection becomes $O(p^2)$. Even if we would distribute runtime deadlock detection with $O(p)$, our overhead would still scale linearly, which would render a distributed approach impractical. Thus, we investigate the analysis time per MPI operation closely to provide a foundation for a distributed implementation.

B. Delayed WFG Construction

MUST overcomes Umpire's limitation by constructing the WFG only on demand, i.e., if MUST invokes a deadlock detection. Thus, with p processes, the detector analyzes the wait-for dependencies of up to p operations during a WFG construction. Each operation may require up to p arcs, thus, the WFG construction has a complexity of $O(p^2)$, which matches the cost of the actual deadlock detection. For our goal of runtime deadlock detection with about 10,000 processes this infrequent overhead stays acceptable. The matching and wait-state analysis costs of different types of MPI operations in MUST are:

Send/Receive:	$O(1)$ if receives/sends use individual queues per communicator, send-receive rank pair, and tag;
Wildcard receive:	$O(p)$ to search all processes for a matching send call;
Single completion:	$O(\log k)$ to check the matching state of the nonblocking communication that the request refers to, which we find in $O(\log k)$ for k requests;
Multi completion:	$O(n \log k)$ to check the matching state that is associated with each of the n request in the completion;
Collective operation:	$O(1)$ to check whether all processes have issued the collective operation.

Analyzing send, receive (with specified source), and collective calls requires $O(1)$, which allows our detector to handle them with low overhead. The detector can also analyze completion calls that use a single request efficiently with $O(\log k)$. The challenging calls are wildcard receives, with $O(p)$ complexity and multi-completions (i.e., `MPI_Waitall`) with $O(n \log k)$ complexity. Multi-completions require n preceding calls to a send or receive initiator, completions that complete all requests like `MPI_Waitall`, lead to an acceptable complexity of $O(\log k)$ per operation in average (n times $O(1)$ and once $O(n \log k)$). The calls `MPI_Waitsome` and `MPI_Waitany` can impose a higher cost however, as they may only complete one or a few requests.

We show in Section VII that the delayed WFG construction leads to an analysis time that grows with $O(\log p)$ or better for a wide range of applications. This property motivates future distributed implementations of our runtime deadlock detection.

VI. Wildcard Receive Handling

Wildcard receives complicate the correctness and efficiency of runtime deadlock detection. If the detector handles a wildcard call that has no yet known matching send call, it can treat the operation as blocked and uses the OR semantic to model wait-for dependencies. Handling wildcard receives for which multiple matching send calls are available is much more complex, though. MUST needs to adapt its matching decisions to the same decisions that the MPI implementation makes. Otherwise, the detector would not follow the same interleaving as the application run, which will lead to an erroneous analysis. Approaches such as ISP or DAMPI do not suffer from this property, as they rewrite nondeterministic MPI calls such that they enforce a known and controlled interleaving, which turns each execution of the application into a deterministic run.

In order to adapt to nondeterministic choices of the MPI implementation, we need to monitor the `MPI_Status.MPI_SOURCE` field for wildcard receives. Blocking receive calls provide this information when they return, whereas for nonblocking receives the wait or test call

Process 0	Process 1	Process 2
MPI_Send(to:1)	MPI_Irecv(from:ANY, &reqs[0])	MPI_Send(to:1)
	MPI_Irecv(from:2, &reqs[1])	
	MPI_Waitall(2, reqs)	
MPI_Barrier()	MPI_Barrier()	MPI_Barrier()

Fig. 7. Schedule-dependent MPI deadlock with an unavailable wildcard completion source.

Process 0	Process 1	Process 2
Irecv(from:ANY, &req)	Isend(to:0, &req)	Isend(to:0, &req)
//long communication	*//long communication*	*//long communication*
Wait(req)	Wait(req)	Wait(req)

Fig. 8. Late wildcard receive completion.

that completes the communication provides this information. Although the detector may determine that a nonblocking receive can match some send, the detector must still wait until the application completes the receive. The detector uses this information to choose the same match as the MPI implementation selected. Thus, the detector must queue all MPI operations that the application issues until the completion result arrives. Due to this queuing, existing tools, such as Umpire [7], can exhibit undesirable behavior in the following three scenarios:

- A deadlock occurs before or while the application executes a completion;
- The application never executes the completion; and
- The detector requires more memory than available to queue MPI operations.

These scenarios lead to unresponsiveness or incomplete analyses, which can render the tool incapable of detecting some deadlocks. We introduce advanced wildcard handling in MUST that overcomes these limitations. We still pause the analysis of MPI operations when we wait for the completion of a nonblocking wildcard receive, but we add two new analysis modes: *probing* and *deciding*. These modes handle the cases that could lead to tool unresponsiveness or incomplete analyses.

A. Probing in MUST

We use *probing* when a timeout invokes deadlock detection while a wildcard receive with at least one known match was not yet completed. Fig. 7 illustrates a situation that largely resembles the scenario in Fig. 1(d). If the first call to `MPI_Irecv` from process 1 matches the send of process 0, the application will complete. Otherwise, this example deadlocks with processes 0 and 2 being stuck in the `MPI_Barrier` call (assuming that the MPI implementation buffers the `MPI_Send` call of process 0) and process 1 in the `MPI_Waitall` call. Since the completion call (in this case `MPI_Waitall`) hangs, the tool cannot obtain the matching decision from the MPI implementation. Thus, we must detect the deadlock without this information.

The *probing* mode matches each wildcard receive with all matching send calls, in order to determine if a deadlock exists. If a specific matching decision leads to deadlock, then we report the error and abort any further analysis. Otherwise, we repeat our *probing* process until we have tested all possible matches. If no matching decision leads to deadlock then the detector waits for additional MPI operations.

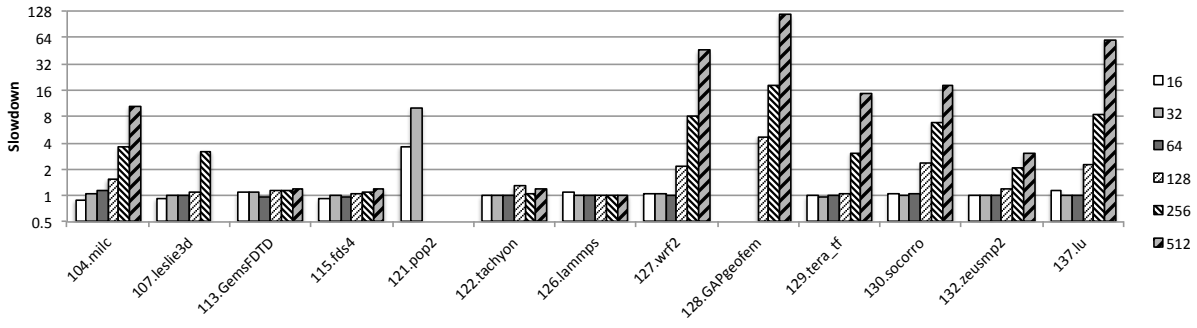
We give an example for this approach in Fig. 7. Process 2 encounters a wild card receive operation. MUST then tests for matches both with the send in process 0, which does not lead to a deadlock, followed by a test with the send in process 2, which reveals the deadlock.

Several synthetic tests of Umpire and MUST require *probing*. In particular, certain stress tests require MUST to investigate multiple hundred interleavings in order to detect a deadlock. In theory, the number of available interleavings can be exponential. However, we are not aware of any cases where only one (or a few) of such an exponential number of interleavings lead to deadlock.

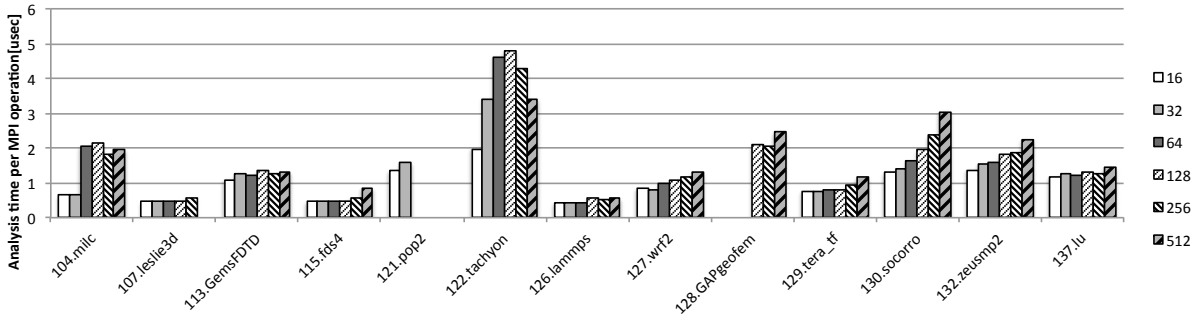
We suspend the search of possible matches after a configurable time period and restart the search with an increased time period if no additional MPI operations arrive. If the search space exceeds the space that MUST can cover in a given amount of time, we might not be able to report a deadlock (false negative). MUST notifies the user if probing starts and allows users to complete the wildcard receives in question at an early time, in order to avoid this rare situation all together.

B. Deciding in MUST

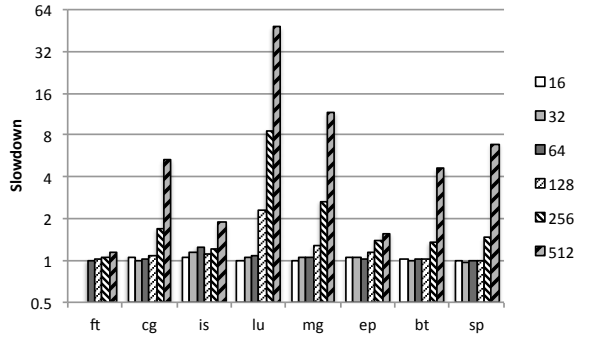
When MUST waits for a wildcard source, the detector queues all other MPI operations until the source arrives. If these queues grow too long to fit into memory, we use the *deciding* mode. Fig. 8 illustrates such a scenario where a nonblocking wildcard receive of process 0 has multiple available matches, while a completion only occurs after a long phase of additional communication events. MUST will start *deciding* if the amount of memory to store these events would exceed the available main memory. MUST first performs a *probing* step to determine if a deadlock exists. If so, we report the error and abort. Otherwise, we enforce some matching decision to allow the execution to continue. In the example in Fig. 8, MUST might decide to match the receive of process 0 with the send of process 1. This measure can cause MUST to report false positives, so we note this behavior in our correctness log and allow users to issue a completion at an earlier time or to



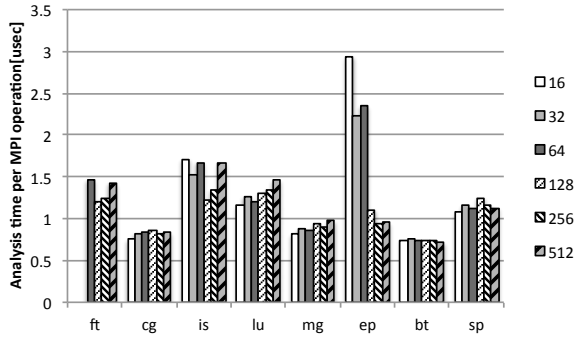
(a) Slowdown for SPEC MPI2007.



(b) Analysis time per MPI operation for SPEC MPI2007.



(c) Slowdown for NPB.



(d) Analysis time per MPI operation for NPB.

Fig. 9. MUST overheads and associated analysis time per MPI operation.

add a missing completion. Applications for which MUST can report false positives are rare, as their communication pattern must be dependent on wildcard matching decisions. Note that approaches such as DAMPI do not support such applications at all.

VII. Application Results

We use version 3.3 of the well known NAS Parallel Benchmarks (NPB) [14] and version 2.0 of SPEC MPI2007 [15] to evaluate MUST's improvements in runtime deadlock detection. We use NPB problem size *D* and the *mref* size for SPEC MPI2007. We run these benchmarks on a Linux-based cluster with $1,944$ nodes of two 6 core Xeon 5660 processors. Each node has 24 GB of main memory and uses a QDR InfiniBand interconnect. We use a range of 16 to 512 cores to measure the behavior of MUST for increased scale and to validate the improvements of our optimized detector implementation. The NPB kernels *bt* and *sp* require square numbers of processes so we use 36 instead of 32, 121 instead of 128, and 529 instead of 512 processes for them. For simplicity, we do not highlight this difference in our graphs. Further, the kernel *126.lammps* contains a potential send-send deadlock that MUST detects. In this case we measure MUST's overheads and analysis time per MPI operation for all operations that MUST analyzes to detect the deadlock.

MUST currently offers three operation modes of communication between the MPI processes and the master: The first uses synchronous communication, the second uses immediate asynchronous communication, and the third uses aggregated asynchronous communication. The synchronous mode only issues an MPI call on the application processes after the

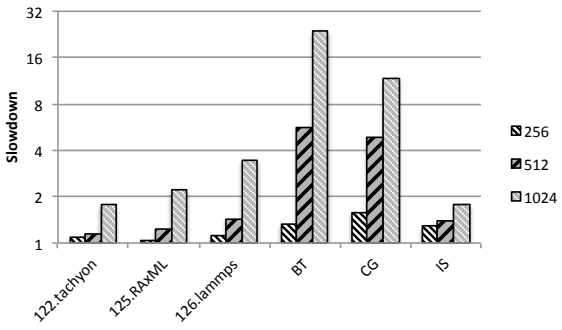


Fig. 10. MUST overheads with larger data sets.

centralized detector analyzed the respective event, which reflects the operation mode of Marmot. The second mode immediately starts a non-blocking communication to notify the detector of a new communication event, whereas the third mode aggregates multiple events into a larger contiguous buffer for higher bandwidth communication. Umpire uses the last mode in combination with errorhandlers, signal handlers, and *atexit* handlers to guarantee that the centralized detector can process all important events even if an application process crashes or hangs. MUST also implements this communication mechanism, but our crash handling still operates at a prototype state where we need additional investigation to guarantee that we catch all bugs on a wide range of platforms. We use the aggregated communication strategy for our measurements and provide the other communication strategies for cases where our crash handling might fail.

Figs. 9(a) and 9(c) show application slowdowns with MUST; a slowdown of 2 corresponds to a 100% increase in application run time. The MUST overhead includes the run time for all the correctness checks that we provide, as well as run times that MUST's communication system consumes to forward MPI call information to the central deadlock detector. We run local correctness checks such as invalid argument checks or MPI resource misuse detection directly on the application processes. An additional MPI process executes the central detector that checks type matching, verifies collective operations, and runs our deadlock detection.

We can analyze all kernels except *107.leslie3d* and *121.pop2* at 512 processes. MUST can handle most kernels with a slowdown of about 2 or lower at 128 processes. The particularly challenging benchmark *121.pop2* invokes an extremely high number of point-to-point calls (about 50,000 per process per second at 32 processes already).

The scalability limit of NPB problem size *D* and of the *mref* input set for SPEC MPI2007 is at about 512 processes, which increases MUST's overhead due to a high communication to computation ratio. Fig. 10 shows MUST's overhead for selected NPB kernels at problem size *E* and for the *lref* input set for SPEC MPI2007. The chart shows that MUST can handle applications at 1024 processes.

Figs. 9(b) and 9(d) show the analysis time per MPI operation of our deadlock detector. As mentioned in Section V, this metric indicates whether distributed runtime deadlock detection is feasible or not. If this time increases linear with scale, a distributed detection would be impractical. For all benchmarks our optimizations in MUST lead to constant or logarithmic increases in the analysis time per MPI operation. Only the benchmark *130.soccorro* shows a slight deviation from this rule. The benchmark completes arrays of MPI requests with repeated calls to `MPI_Waitany`, which likely causes this behavior. As discussed in Section V, this behavior may lead to a linear complexity.

VIII. Conclusions

We present MUST, a novel runtime error detection tool for MPI applications. Key features include type matching, collective verification, and deadlock detection. We contribute theoretic and processing extensions for the AND⊕OR model based deadlock detection. With p processes, the existing approach based on this model required an analysis time of $O(p)$ for each blocking MPI operation, which makes deadlock detection prohibitively expensive at scale. We overcome this limitation and achieve an analysis time of $O(\log p)$ per MPI operation or better for actual applications. We demonstrate this result for two major benchmark suites for up to 1024 processes. Additionally, the generalization of our deadlock model allows us to handle complex wait-for semantics that arise with certain existing MPI constructs and will become more common with future MPI extensions such as nonblocking collectives.

Although our current approach can scale to at least 1024 processes for some applications and inputs, future use cases will need additional advances. Deadlock detection becomes more challenging as systems scale even further and some errors may only occur at higher scales. Our work provides a basis for scalable distributed MPI runtime deadlock detection, but we will need to develop additional scalable techniques for point-to-point matching, collective matching, and wait-state analysis to do so.

Acknowledgments

Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. (LLNL-CONF-555978). This work has been supported by the CRESTA project that has received funding from the European Community's Seventh Framework Programme (ICT-2011.9.13) under Grant Agreement no. 287703.

References

[1] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard, Version 2.2," http://www.mpi-forum.org/docs/mpi22-report.pdf, April 2009.

[2] T. Hilbrich, B. R. de Supinski, M. Schulz, and M. S. Müller, "A Graph Based Approach for MPI Deadlock Detection," in *ICS '09: Proceedings of the 23rd international conference on Supercomputing*. New York, NY, USA: ACM, 2009, pp. 296–305.

[3] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby, "ISP: A Tool for Model Checking MPI Programs," in *13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008, pp. 285–286.

[4] G. R. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou, "MPI-CHECK: A Tool for Checking Fortran 90 MPI Programs," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 2, pp. 93–100, 2003.

[5] C. Falzone, A. Chan, and E. Lusk, "Collective Error Detection for MPI Collective Operations," in *In Recent Advances in Parallel Virtual Machine and Message Passing Interface, 12th European PVM/MPI Users' Group Meeting*. Springer, 2005, pp. 138–147.

[6] B. Krammer and M. S. Müller, "MPI Application Development with MARMOT," in *PARCO*, ser. John von Neumann Institute for Computing Series, vol. 33. Central Institute for Applied Mathematics, Jülich, Germany, 2005, pp. 893–900.

[7] J. S. Vetter and B. R. de Supinski, "Dynamic Software Testing of MPI Applications with Umpire," *Supercomputing, ACM/IEEE 2000 Conference*, pp. 51–51, 04-10 Nov. 2000.

[8] A. Vo, "Scalable Formal Dynamic Verification of MPI Programs through Distributed Causality Tracking," Ph.D. dissertation, University of Utah, School of Computing, March 2011.

[9] V. C. Barbosa and M. R. F. Benevides, "A Graph-Theoretic Characterization of AND-OR Deadlocks," 1998.

[10] S. Lee, "Fast, Centralized Detection and Resolution of Distributed Deadlocks in the Generalized Model," *IEEE Trans. Softw. Eng.*, vol. 30, no. 9, pp. 561–573, 2004.

[11] H. J. Yoon and D. Y. Lee, "Deadlock-Free Scheduling of Photolithography Equipment in Semiconductor Fabrication," *Semiconductor Manufacturing, IEEE Transactions on*, vol. 17, no. 1, pp. 42–54, Feb. 2004.

[12] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI," in *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM, Nov. 2007.

[13] T. Hilbrich, M. S. Müller, B. R. de Supinski, M. Schulz, and W. E. Nagel, "GTI: A Generic Tools Infrastructure for Event Based Tools in Parallel Systems," in *To appear in IPDPS 2012: Procedings of the 26th IEEE International Parallel & Distributed Processing Symposium*, 2012.

[14] D. H. Bailey, L. Dagum, E. Barszcz, and H. D. Simon, "NAS Parallel Benchmark Results," IEEE Parallel and Distributed Technology, Tech. Rep., 1992.

[15] "SPEC MPI2007 Benchmark Suite for MPI," http://www.spec.org/mpi2007/.